Predicting and Analyzing GCC Optimization Levels for Performance of C Programs

Akshay Gopalakrishnan

McGIll Id**: 260849432**

Department of Computer Science

McGill University

akshay.akshay@mail.mcgill.ca

GitHub: https://github.com/jaag5678/CompOptML.git

Abstract

Compiler optimizations transform a given program to an equivalent form that performs better that the original program. Performance can be with respect to execution time, memory, compilation time, power consumption, etc. Each optimization has its own impact on the individual performance metrics mentioned above. There are ample of optimizations out there and based on the performance required different optimizations are used. The order in which these optimizations are used has a great impact of the resultant equivalent code generated. In order to ensure a proper order, current systems adopt a method called Iterative Compilation mentioned in [1] which does a brute force search on all orders of optimizations available. On doing so, getting a good order is a cakewalk. Another way to look at this problem is to view a program as a state and choose the optimization that transforms a program to another state, which is one step closer to the best state of program. This project attempts to deal with a subset of this problem, that is, given a program what are the set of optimizations that should be done to improve performance. We implement this approach for C programs for which we predict which GCC optimization level would be best to improve a sub metric of performance, which is execution time.

Introduction

Compiler optimization have been of interest to computer scientists for quite a few decades. The ability to have a computer transform our programmed ideas into better form that supersedes performance is something that has always intrigued language developers. This is mainly because many programs have the same objective and if a machine could understand that and transform it to a similar better performing code, a lot of work can be saved. Computer scientists over time have come up with hundreds of optimizations that transform a program in different ways based on specific traits of a program analyzed. One of the key problems in the research on optimization is the ability of the system to understand the semantics of a program. This task in itself is intractable. Developers hence focus on refining other aspects of optimization. One such problem is the decision to order the hundreds of optimization for a given program. Optimizations done in different order would lead to different equivalent forms of programs which may not be the best in terms of performance. This is usually termed as the phase ordering problem as mentioned in [2].

Approaches using machine learning, view the phase ordering problem as a Markov decision process, wherein a program is a state, and an optimization changes the state of the program. The objective of the phase ordering problem then becomes choosing the path that leads to the final state of the program whose performance supersedes previous states. Given hundreds of optimizations out there, my focus lies in the subset of this task, that is, given a program state, which are the best set of optimizations that should be done.

Project Description

Optimizations for programs are intended for specific improvement in certain performance metrics such as execution time, compilation time, memory used, power consumption, etc. For this project I focus on just the optimization that improves the execution time of programs.

Given hundreds of optimizations out there for different programming languages, I decided to first decide the language whose optimizations would be of my interest. Due to my familiarity with C programs, I decided to choose

this language. Furthermore, C language itself has hundreds of optimizations out there, most of which are very specific to certain programs. For this reason, I decided to predict a group of optimizations that should be done.

Thus, this project aims to predict set of optimizations for a given C program. To be more specific, the set of optimizations that I would deal with is the O2 and O3 levels given by the GCC compiler for C programs which are given in [3] .

This project is mainly meant for learning purpose which achieves the following learning goals:

- How to choose dataset for optimization using machine learning
- What are the possible ways to define features for a collection of programs
- What kind of learning realm does the phase ordering problem fall under
- What factors would affect the choice of the machine learning model adopted
- How to analyze and interpret the results of the predictions given by the model itself

To start with this project, I decided to divided the task into four phases:

Phase 1: Searching for an appropriate dataset

Phase 2: Creating labels for the dataset obtained

Phase 3: Defining features for my dataset

Phase 4: Modelling the data given features and labels

The last unsaid phase would be analysis/ observations in each phase, which will be described under each phase itself.

Phase 1: Dataset

Every machine learning problem requires a dataset using which we can learn and then use that knowledge to predict in future. Similarly, for our problem, we would need a dataset consisting of C programs. For this I decided to explore existing C benchmark repositories mentioned by [4].

Given the complexity and variety of C programs out there, I decided to search for a dataset with single threaded programs which were not necessarily complex. The reason behind this was to allow analysis of the

predictions of the model to be restricted to a closed group of features common to all C programs in the dataset.

In short I need C programs that could be:

- Labelled as per the 4 levels of optimizations given by GCC
- Simple and consist of a finite set of common traits of C programs
- One program file per code mapped to one label
- Single threaded C codes (no thread libraries involved)

C Benchmark codes:

Each benchmark code has various types of C programs with different implementations of great complexity. In summary:

- They have multiple codes that are linked together to execute them.
- Have too many additional library dependencies
- They are too complex having many C program traits that vary from code to code.
- Multithreaded programs

Need for Dataset Creation:

Having failed to have obtained a suitable benchmark for my dataset, I decided to make my own dataset from my own programs coded on Hackerrank[5]

- The codes are not split across multiple files that need to be linked to execute
- The codes are related to Mathematical problems which involve a fixed set of traits of C code that are common across almost all codes. (eg: Recursion, Square root, Loops)
- The codes do not need additional libraries apart from <stdio.h> and <math.h>
- They are all computationally intensive (except one or two).

Currently, I have gathered around 65 programs for creating a labelled dataset. Each of these codes have been tweaked a little to have a predetermined amount of iterations for major loops in the program. They have an input file from which input is read and appropriate computations are done to provide results. The dataset repository is in my Github repository.

Phase 2: Dataset Labelling

Once the set of programs were gathered, they were compiled using optimization levels [O0, O1, O2, O3] using a bash script which compiled them using the template of gcc command as given below:

gcc –static [optimization level] [program name] -o [ouputfile name] -lm

The version of gcc used was 5.5.0 released on 2017/10/10. For each level of optimization, a separate output file was created.

To measure proper execution time of my programs, the following major issues need to be taken care of:

- Multiple processes should not be running on the system at once, as execution time cannot be measured accurately due to frequent context switches
- The program must run on a single processor.
- If there are too many cache hits / cache misses, the execution time could drastically be affected.

`In order to further reduce any bias in execution time, I ran the programs under different optimizations over 100 iterations and took the average execution time.

The system I used to measure the above has the following specifications:

RAM: 11.7 GB **Operating System:** Ubuntu 16.04 LTS 64-bit

Processor: Intel Core i7-4770 CPU @ 3.40GHz * 8

Phase 3: Defining Features

For my approach, I would require a fixed set of features for every program. There are several ways to do this, as initially I thought to just take counts of specific traits decided by me for every program and my count vector would be my feature for that program.

The above approach however, eliminates the semantics of the program, which is one of the key aspects while optimizing a code. It appears that determining the feature set is very hard indeed. Why not then let a mathematical model decide the features for us??

I could use an autoencoder to encode all my programs into a compact set of encoded features. This could help gaining a fixed set of features for various size of codes. The major assumption here is that the autoencoder will be able

to retain some amount of semantics of the code itself in its encoded form. This is also subject to further analysis.

I implemented an autoencoder using Keras library with a Tensorflow Backend. I referred to the guide given by [6], to get a fair idea of how to implement it using python. For simplicity sake, I used an autoencoder with just one hidden layer, which is the encoding layer. Here are the specifications of the autoencoder used:

Compiling Model:

- Input size = 3769, Output size = 3769
- Encoding dim = 100
- Optimizer used – Adaptive Gradient Descent:
- Loss Function – Mean squared error
- Encoding activation function = ReLu
- Decoding Activation function = Sigmoid

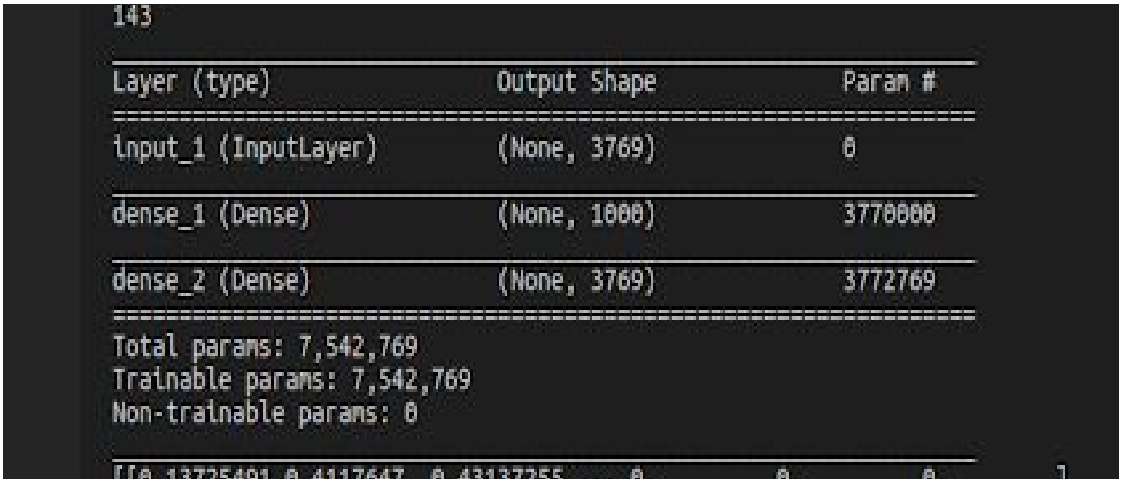
The max size of code had 3769 characters and the least size had 146 characters. Each of the codes were converted to ASCII format, followed by normalizing them for using as training data for our autoencoder. Adaptive gradient descent alters the learning rate adaptively, hence preventing redundant epochs where no improvement takes place. Mean squared error is used to measure the reconstruction loss between input and the output which should be the normalized format of the input code. This loss function would be effective in measuring how accurately the program is regenerated as the absolute difference between every character is squared and summed up to measure the loss. The encoding and decoding activation functions are set to what is used in common practice as mentioned in [6].

Fitting Autoencoder to Data:

- Input, Output = ASCII normalized programs
- Epochs = 50
- Validation split = 0.2 (how much portion of data to be used for validation in each epoch)
- Shuffle = True (randomly shuffle the data for each epoch and feed it to the autoencoder)

For each epoch, the mean squared error is computed along with the validation error. Additionally, the accuracy of each of the above loss metrics is also measured and displayed.

Below is the summary of the number of parameters for the model.



The autoencoder implementation by Keras randomly initializes initial weights to the parameters every time it is run. This could cause certain randomness in our encoded result as the algorithm may converge to different local minima (could also be global at some instance). The average final statistics on running the autoencoder twelve times is given below

Average mean squared error = 0.00245

Average validation error = 0.003133

Phase 4: Modelling data and Prediction

In practice the most common optimization levels used are O2 or O3; so for starters, I plan to predict for a given code whether to choose O2 or O3 as the set of optimization to improve execution time. This boils down to a simple binary classification problem. A logistic regression approach was taken using python libraries to model this binary classification. The algorithm was implemented using the Logistic Regression model in[7]. Here are the brief details of implementation:

- Dataset size = 65
- Training size = 52 Test size = 13
- KFold Cross-validation used where K = 5
- Solver for Logistic Regression used was Liblinear
- Tolerance = 1e-5

For each of the autoencoder output of the encoded format, the logistic model was trained. The model was however tested on all the data to measure how well the model performs. For each of the results, I created two metrics to observe and evaluate the model:

1. **True Classification – Predicted Classification**

Since my labels for O2 and O3 were 2 and 3 respectively, the difference would give me insight as to how many are predicted right. A difference of:

- 0 = Accurate prediction
- 1 = Predicted O2 instead of O3
- -1 = Predicted O3 instead of O2

2. **Predicted Probabilities for each class:**

The predicted probabilities of each class for each data input would tell us how confident our model is in predicting a given class. Note that the sum of the probabilities would be one. Which makes it an important factor to assess the performance of the model.

3. **Score Accuracy:**

A measure of how accurate the labels are predicted. This is just the fraction of labels predicted right over total size of the training data.

Results

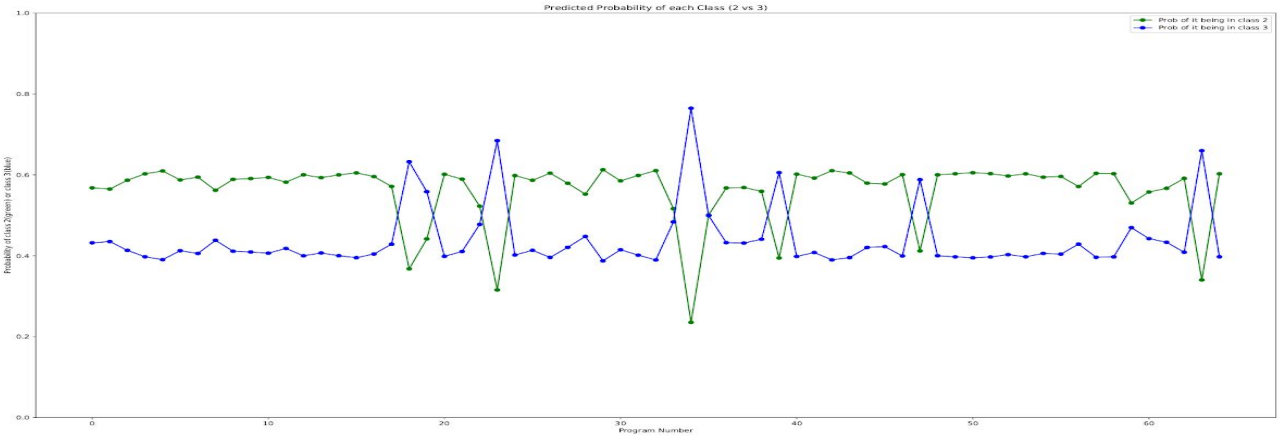
For each run of the autoencoder, we get a unique set of encoded format of the program. This is due to the random initialization of weights while training the autoencoder. For the logistic regression model, it could be kept initialized to be one whenever it starts. So the results would be the same with respect to that.

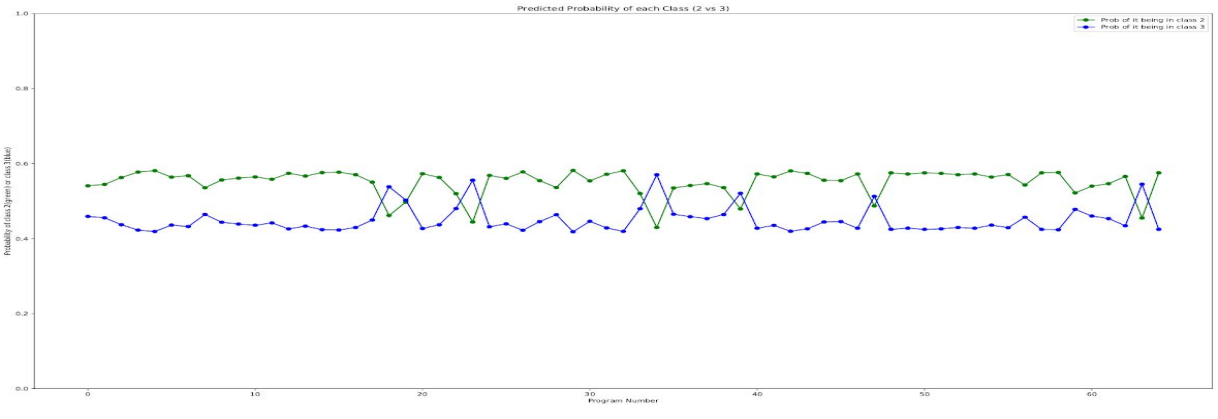
Twelve tests were done right from training the autoencoder to training the logistic model. The results are as follows.

Test	MeanSqError (Auto Encoder)	Validation Error (Auto Encoder)	Number of mispredictions	Accuracy (score)	Misprection Split (1, -1, 0)
1	0.0027	0.0034	23	0.653846	22, 1, 42
2	0.0023	0.0031	23	0.653846	22, 1, 42
3	0.0022	0.0030	23	0.653846	22, 1, 42
4	0.0023	0.0030	23	0.653846	22, 1, 42
5	0.0023	0.0031	23	0.653846	22, 1, 42
6	0.0026	0.0032	24	0.634615	24, 0, 41
7	0.0022	0.0031	24	0.634615	23, 1, 41
8	0.0025	0.0033	23	0.653846	22, 1, 41
9	0.0023	0.0031	23	0.653846	22, 1, 41
10	0.0023	0.0030	6	~1.0	3, 3, 59
11	0.0035	0.0033	24	0.63461	22, 2, 41
12	0.0022	0.0030	4	~1.0	3, 1, 61

The columns have their meanings as per defined above in the previous sections. Misprediction split is as per the True Classification – Predicted Classification metric, which counts the number of misprediction for O2 and O3 classes followed by correct prediction.

Although the Accuracy metric is common for most of them, their corresponding graphs of predicted probabilities differ quite a lot. As an example, the graphs for predicted probabilities for Test 2 and 3 are as follows:

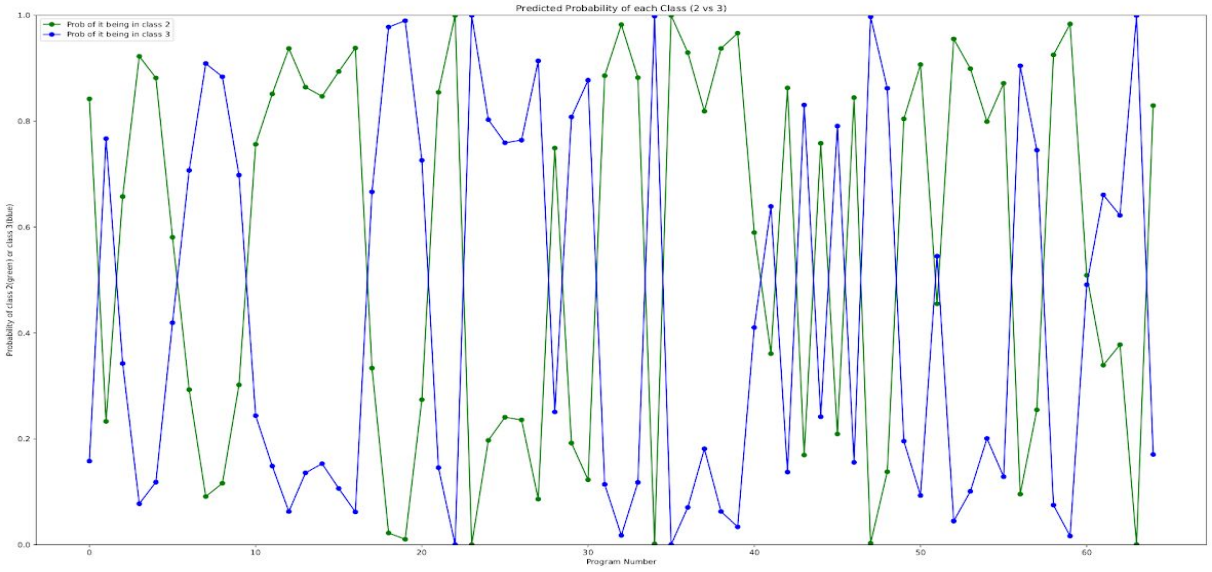




In case the legend is not visible for both, the green line indicated probability of the program being suggested O2 optimization level and the blue line indicates the probability of the program being suggested O3 optimization level. The X axis represents the program number and the Y axis specifies the value of probability.

As you can see, although having the same accuracy score, the probabilities are entirely different. This is mostly due to the randomization of weights that is being done while training the autoencoder itself, which in turn results in the optimization objective to converge at some local minima. This is not due to the logistic model training as the weights are initialized to be to be 1 by default as mentioned by [7].

Another interesting thing to notice are test 10 and 12, which gives almost 100% accuracy. This can be attributed again to the random initialization of weights during autoencoder training phase. The accuracy however could mainly be due to overfitting as my dataset is only a collection of 65 programs. However it could also be that the encoded representation is actually capturing some level of semantics of the program when converging to some local minima and this is an interesting result that would require more further analysis. If the latter is true, then this would also mean that the decision boundary is linear itself, and that no non linear factors would need to be considered while predicting between O2 and O3. However, given my limited traits of programs and their count which is 65 this would be highly unlikely.



The similar graph for Test 10 which shows the predicted probabilities also show that the model is confident about its prediction. By confident, I mean to say the probability of one of the class prediction for given program is close to 0.8 and the other close to 0.2. The graph is shown above.

In conclusion, I would like to say that identifying the correct set of initializations of weights for autoencoder would allow us to give good results. But having a deterministic method for initialization seems infeasible. Additionally, due to limited dataset, the model predictions itself may not be that accurate. But assuming this is not an issue, it would be safe to say that on an average the model has accuracy of predicting 65% of the samples.

NOTE: All the graphs are in my github repository, under the Results Folder so please refer the entire 12 test results there, also if graph not legible please refer it from my github.

Future Work

Getting more samples for the dataset is probably the most important thing for this project to get more concrete results that could be analyzed. However, it is also known that amount of data does not always result in giving good performance or allow us to analyze the results better.

Having said that, there is also scope in changing and trying various optimizers, loss functions, and other hyperparameters of the autoencoder to try and see which of them give better results. Perhaps, using Bayesian Optimizations to choose the optimal hyperparameters for the Autoencoder would be a good thing to do in future. Additionally, trying to model the data using other forms of machine learning models would also be worth exploring.

Prior to using autoencoders, perhaps parsing and extracting relevant tokens in sequence to get a general semantics of each program which could be used as input to the autoencoder is a very promising direction to try capture the semantics of the code using machine learning to predict optimizations.

Using the same process and working on individual optimizations is what I would currently intend to do soon.

An early analysis could be done by clustering the token sequence of the programs based on some similarity metric such as subsequence or a metric mentioned in [8] . This encourages applying unsupervised learning to analyze optimizations that could be applied to a program for a better performance in future. Perhaps combining both supervised and unsupervised learning results and getting a sense of what each model is capturing would be

vital in understanding whether machine learning can really be used for the phase ordering problem.

Conclusion

An attempt to use machine learning to predict set of optimizations was done. The dataset for this was created to ensure that every program has a limited set of traits that could be further used to analyze what the machine learning model learns. The features for each program was generated using an autoencoder, encoding every program to 100 features. These features were then used to train a logistic regression model which was then tested for its accuracy. It was concluded that the average accuracy was around 65%. Nothing conclusive could be said about whether this process could capture any level of semantics. Further analysis on the hyperparameters of the model as well as gathering more data is something that I would consider most important for immediate future work.

References

[1]Ashouri, Amir Hossein, William Killian, John Cavazos, Gianluca Palermo and Cristina Silvano. “A Survey on Compiler Autotuning using Machine Learning.” *CoRR*abs/1801.04405 (2018): n. pag.

[2] Ashouri, Amir Hossein, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni and John Cavazos. “MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning.” *TACO* 14 (2017): 29:1-29:28.

[3]https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[4]http://www.nkavvadias.com/benchmarks.html

[5]https://www.hackerrank.com/dashboard

[6]https://blog.keras.io/building-autoencoders-in-keras.html

[7]https://scikitlearn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html#

[8]Luiz G.A. Martins, Ricardo Nobre, Alexandre C.B. Delbem, Eduardo Marques, and João M.P. Cardoso.2014. Exploration of compiler optimization sequences using clustering-based selection. SIGPLAN Not. 49,5 (June 2014), 63-72. DOI: https://doi.org/10.1145/2666357.2597821